The Essence of Form Abstraction

Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop

School of Informatics, University of Edinburgh

Abstract. Abstraction is the cornerstone of high-level programming; HTML forms are the principal medium of web interaction. However, most web programming environments do not support abstraction of form components, leading to a lack of compositionality. Using a semantics based on idioms, we show how to support compositional form construction and give a convenient syntax.

1 Introduction

Say you want to present users with an HTML form for entering a pair of dates (such as an arrival and departure date for booking a hotel). In your initial design, a date is represented just as a single text field. Later, you choose to replace each date by a pair of pulldown menus, one to select a month and one to select a day.

In typical web frameworks, such a change will require widespread modifications to the code. Under the first design, the HTML form will contain *two text fields*, and the code that handles the response will need to extract and parse the text entered in each field to yield a pair of values of an appropriate type, say, an abstract date type. Under the second design, however, the HTML will contain *four menus*, and the code that handles the response will need to extract the choices for each menu and combine them in pairs to yield each date.

How can we structure a program so that it is isolated from this choice? We want to capture the notion of *a part of a form*, specifically a part for collecting values of a given type or purpose; we call such an abstraction a *formlet*. The designer of the formlet should choose the HTML presentation, and decide how to process the input into a date value. Clients of the formlet should be insulated from the choice of HTML presentation, and also from the calculation that yields the abstract value. And, of course, we should be able to compose formlets to build larger formlets.

Once described, this sort of abstraction seems obvious and necessary. But remarkably few web frameworks support it. Three existing web programming frameworks that do support some degree of abstraction over form components are WASH [26], iData [22] and WUI [10, 11], each having distinctive features and limitations. (We discuss these further in Section 7.) Our contribution is to reduce form abstraction to its essence. Defining a semantics for formlets by composing primitive *idioms* [18], we show how to support compositional form construction, and give a convenient syntax. Furthermore, we illustrate how the semantics can be extended to support additional features either by composing with additional primitive idioms or by generalising to indexed and parameterised idioms.

The difficulty arises from the fact that the standard HTML interface is rather low level. Amongst the troubles it poses:

- There is no static association between a form definition and the code that handles it, so the interface is fragile. This means the form and the handling code need to be kept manually in sync.
- Field values are always received individually and always as strings: HTML and CGI provide no facility for processing data or giving it structure.
- Given two forms, there is generally no easy way to combine them into a new form without fear of name clashes amongst the fields—thus it is not easy to write a form that abstractly uses subcomponents. In particular, it's difficult to use a form twice within a larger form.

Conventional web programming frameworks such as PHP [21] and Ruby on Rails [23] facilitate abstraction only through templating or textual substitution, hence there is no automatic way to generate fresh field names, and any form "abstraction" (such as a template) still exposes the programmer to the concrete field names used in the form. Even advanced systems such as PLT Scheme [9], JWIG [5], scriptlets [8], Ocsigen [2], Lift [14] and the original design for Links [6] all fall short in the same way. The situation begs for an abstraction on top of HTML forms.

Formlets address all of the above problems: they provide a static association between a form and its handler (ensuring that fields referenced actually exist and are of the right type), they allow processing raw form data into structured values, and they allow composition, in part by generating fresh field names at runtime.

Most interesting, the formlet abstraction turns out to be an *idiom* [18], also known as *applicative functor* or *lax monoidal functor*, (a notion of effectful computation, related to both monads [19] and arrows [13]), and further it is expressible as the composition of three standard idioms. In this paper we give such a definition in OCaml, which also comprises a complete working implementation. We take advantage of the extensible Camlp4 preprocessor to provide syntactic sugar, without which formlets are usable but more difficult to read and write. Both the library and the syntax extension are available from

`http://groups.inf.ed.ac.uk/links/formlets/`

Formlets are also implemented in Links, which also provides the syntax presented here. The complete Links system includes many features, such as a full suite of HTML controls (textareas, pop-up menus, radio buttons, etc.), which are not described here. Steve Strugnell has ported a commercial web-based project-management application originally implemented in PHP to the Links version of formlets [24]. He gives an in-depth comparison between Links formlets and forms implemented in PHP.

The remainder of this paper is organised as follows. Section 2 lays out the background of HTML forms and the difficulties in abstracting them. Section 3

presents formlets, as they appear to the programmer, through examples. Section 4 gives a semantics for formlets based on the composition of the three idiom instances which capture the effects needed for form abstraction. Section 5 defines formally the formlet syntax used throughout the paper and relates it to the formlet idiom. Section 6 shows how to extend the basic abstraction with additional features: static XHTML validation, user-input validation, and an optimised representation based on multi-holed contexts. Section 7 examines the relationship with existing form-abstraction features in high-level web frameworks.

2 Abstracting form components

2.1 HTML forms and CGI

When a user submits an HTML form, the structure of the submission can be viewed as a flat list of field-value pairs, both strings. This flatness is embedded in both the HTML specification, which defines how the form is specified, and the CGI specification, which defines the application's interface to the submitted data. It is common to view the submission as an association list or a function that maps field names to their string values.

Meanwhile, the presentational structure of an HTML form is far from flat: it can consist of a variety of nested HTML elements that may arbitrarily nest the individual input controls.

Together, these are the two aspects of HTML forms which we are interested in manipulating: the presentational structure and the ultimate *outcome* or *submitted data*.

2.2 Abstracting HTML forms: Requirements

In view of these two form aspects (presentation and outcome), we aim to abstract as completely as possible over both. Let us consider what this means.

First, though the HTML/CGI interface presents the submitted data as a flat string-to-string mapping, abstractly we can conceive of the form outcome as being in any data type. Arbitrary computation may be needed to calculate a value in that result type.

Next, the abstraction should not entail any particular mode of user interaction, and in particular we make no assumptions about how many times the form will be displayed or submitted. A program *may* display a particular form more than once, but that is up to the designer; what the program does after the form is submitted is up to the programmer. The ultimate result of any form submission is simply the submitted data, which the program is free to treat in any way appropriate.

The abstraction should also permit arbitrary presentational combination: it should be possible to set two components side by side, top-to-bottom, or in any other arrangement that HTML affords.

It should be clear that the abstraction must generally support arbitrary computations on both the presentation structure and on the submitted data. We should be able to write combinators that alter either aspect, or both at once.

3 Formlets by example

Now we illustrate formlets, as they might appear to the programmer, with an example. We assume familiarity with HTML and OCaml. This section covers our OCaml implementation, and so has features that may vary in another implementation of formlets. We use a special syntax (defined formally in Section 5) for programming with formlets; this syntax is part of the implementation, and makes formlets easier to use, but not an essential part of the abstraction.

The following code creates a formlet, called *date*, with two text input fields, labelled "Month" and "Day":

```
let date : date formlet =
  formlet
    <div>
      Month :  {input_int ⇒ month}
      Day :    {input_int ⇒ day}
    </div>
  yields
    make_date month day
```

Upon submission, this formlet will yield a *date* value representing the date entered. The user-defined *make_date* function translates the day and month into a suitable representation.

A formlet expression consists of a *body* and a *yields clause*. The body of the *date* formlet is

```
<div>
  Month :  {input_int ⇒ month}
  Day :    {input_int ⇒ day}
</div>
```

and its yields clause is

```
make_date month day
```

The body of a formlet expression is a *formlet quasiquote*. This is like an XML literal expression but with embedded *formlet bindings*. A formlet binding $\{f \Rightarrow p\}$ binds the value yielded by f to the pattern p in the yields clause. Thus here the variables *month* and *day* will be bound to the values yielded by the two instances of the *input_int* formlet. Because of this binding behavior, we cannot attempt to use the value of a non-existent form field, which traditional CGI interfaces would allow. The place the formlet binding occupies in the quasiquote is replaced, when evaluated, by the rendering of the bound formlet f.

The value *input_int* : *int formlet* is a formlet that renders as an HTML text input element, and parses the submission as type *int*. Although the *input_int* formlet is used here twice, the formlet library ensures that no field name clashes arise.

It is important to realize that any given formlet defines behavior at two distinct points in the program's runtime: first when the form structure is built up, and much later (if at all) when the form is submitted by the user, when the outcome is processed.

Next we illustrate how user-defined formlets can be usefully combined to create larger formlets. We construct a travel formlet which asks for a name, an arrival date, and a departure date.

```
let travel : (string × date × date) formlet =
  formlet
    <#>
      Name:{input ⇒ name}
      <div>
        Arrive:{date ⇒ arrive}
        Depart:{date ⇒ depart}
      </div>
      {submit "Submit"}
    </#>
  yields
    (name, arrive, depart)
```

The *input* formlet simply allows the user to enter a string using an input element. The library function *submit* just evaluates to the HTML for a submit button; its string argument provides the label for the button. (This covers the common case where there is a single button on a form. A similar function *submit_button* : *string* → *bool formlet* constructs a submit button formlet, whose result indicates whether this button was the one that submitted the form.)

Having created a formlet, how do we use it? For a formlet to become a form, we need to connect it with a *handler*, which will consume the form input and perform the rest of the user interaction. and render it as part of a web page. The function *handle* attaches a handler to a formlet.

Continuing the above example, we render *travel* onto a full web page, and attach a handler that displays the chosen itinerary back to the user. (The *xml* type is given in Figure 1; we construct XML using special syntax, which is defined in terms of the *tag* and *text* functions, as shown formally in Section 5.)

```
let display_itinerary : (string × date × date) → xml =
  fun (name, arrive, depart) →
    <html>
      <head><title>Itinerary </title></head>
      <body>
        Itinerary for: {xml_text name}
        Arriving: {date_of_xml arrive}
        Departing: {date_of_xml depart}
      </body>
    </html>

handle travel display_itinerary
```

This is a simple example; a more interesting application might render another form on the *display_itinerary* page, one which allows the user to confirm the itinerary and purchase tickets; it might then take actions such as logging the purchase in a database, and so on.

```
type xml = xml_item list
 and tag = string
 and attrs = (string × string) list
 and xml_item

val xml_tag : tag → attrs → xml → xml
val xml_text : string → xml
```

Fig. 1. The *xml* type.

This example demonstrates the key characteristics of the formlet abstraction: static binding (we cannot fetch the value of a form field that is not in scope), structured results (the month and day fields are packaged into an abstract *date* type, which is all the formlet consumer sees), and composition (we reuse the date formlet twice in the *travel* formlet, without fear of field-name clashes).

4 Semantics

In order to capture the essence of formlets we need to give them a semantics, preferably using a well-understood formalism. Clearly formlets involve side-effects, in the form of name generation and user interaction. Monads [19, 27, 3] provide a standard semantic tool for reasoning about side-effects. It is not difficult to see that there is no monad corresponding to the formlet type. Intuitively, the problem is that a *bind* operation for formlets would have to read some of the input submitted by the user before the formlet had been rendered, which is clearly impossible. (Recall that the type of bind would be $\alpha\ \mathit{formlet} \to (\alpha \to \beta\ \mathit{formlet}) \to \beta\ \mathit{formlet}$ and to implement this would require extracting the α value from the first argument to pass it to the second argument; but the presentation of the $\beta\ \mathit{formlet}$ should not depend on the α-type data submitted to the first formet.)

Idioms are a generalisation of monads that *are* suitable for modelling formlets. A definition of formlets will turn out to be just the composition of three simple, standard idioms, a fact which reveals the essential structure of formlets.

Idioms were introduced by McBride [17] to internalise a common pattern of functional programming. Subsequently McBride and Paterson [18] changed the name to *applicative functor* to emphasise the view of idioms as an "abstract characterisation of an applicative style of effectful programming". We stick with McBride's original "idiom" for brevity.

An idiom is a type constructor I together with operations:

$$\begin{aligned} \mathit{pure} &: \alpha \to \mathrm{I}\,\alpha \\ \otimes &: \mathrm{I}(\alpha \to \beta) \to \mathrm{I}\,\alpha \to \mathrm{I}\,\beta \end{aligned}$$

that satisfy the following laws:

$$\begin{aligned} \mathit{pure}\ \mathit{id} \otimes u &= u \\ \mathit{pure}\ (\circ) \otimes u \otimes v \otimes w &= u \otimes (v \otimes w) \\ \mathit{pure}\ f \otimes \mathit{pure}\ x &= \mathit{pure}(f\ x) \\ u \otimes \mathit{pure}\ x &= \mathit{pure}(\lambda f.f\ x) \otimes u \end{aligned}$$

where *id* is the identity function and $\circ$ denotes function composition.

The *pure* operation lifts a value into an idiom. Like standard function application, idiom application $\otimes$ is left-associative. The idiom laws guarantee that pure computations can be reordered. In particular, an effectful computation cannot depend on the result of a pure computation, and any expression built from *pure* and $\otimes$ can be rewritten in the canonical form:

$$pure\ f \otimes u_1 \otimes \cdots \otimes u_k$$

where f is the pure part of the computation and $u_1, \ldots u_k$ are the effectful parts of the computation. This canonical form captures the essence of idioms as a tool for modelling computation. The intuition is that an idiomatic computation consists of a series of side-effecting computations, each of which returns a value. As with monads, the order in which computations are performed is significant, but unlike monads subsequent computations cannot depend on the values returned by prior computations. The final return value is obtained by aggregating the values returned by each of the side-effecting computations, using a pure function. As Lindley et al [16] put it: *idioms are oblivious*. We note that every monad is an idiom, though of course being oblivious the idiom interface is less powerful (see Lindley et al [16] on the relative expressive power of idioms, arrows and monads).

The *effectful*, *applicative functor* view of idioms is useful for giving a denotational semantics, and for specifying the basic idiom operations. Our core implementation of formlets is based on this view. The more operational *notion of oblivious computation* view of idioms can be more convenient to program with. Our syntactic sugar is based on the oblivious view. (Note that there are analogous denotational and operational views of monads: the monadic bind and return operations give us the denotational view whereas the computational metalanguage, or equivalently Haskell do notation, gives us the operational view.)

4.1 Definition

The OCaml definition of the idiom interface is as follows:

```
module type Idiom = sig
  type α t
  val pure : α → α t
  val (⊗) : (α → β) t → α t → β t
end
```

Figure 2 shows the definition of the formlet idiom.

4.2 Factoring formlets

Now we introduce the three idioms into which the formlet idiom factors. We use a *namer*, which generates fresh form-field names; a *writer* (or monoid-accumulator), which accumulates the form structure as we build it; and a *reader*,

```
module type FORMLET = sig
  include Idiom
  val xml : xml → unit t
  val text : string → unit t
  val tag  : tag → attrs → α t → α t
  val run : α t → xml × (env → α)
  val input : string t
end

module Formlet : FORMLET =
struct
  type α t = int → (xml × (env → α) × int)

  let pure x i = ([], const x, i)
  let (⊗) f p i =
    let x1, g, i = f i in
    let x2, q, i = p i in
      (x1 @ x2, (fun env → g env (q env)), i)

  let xml x i = (x, const (), i)
  let text t i = xml (xml_text t) i
  let tag t attrs fmlt i =
    let x, f, i = fmlt i in
      (xml_tag t attrs x, f, i)

  let next_name i = "input_" ^ string_of_int i, i + 1
  let input i =
    let w, i = next_name i in
      (xml_tag "input" [("name", w)] [], List.assoc w, i)

  let run c = let x, f, _ = c 0 in (x, f)
end
```

Fig. 2. The formlet idiom.

which passes the raw form outcome through the processing code to achieve the ultimate outcome (Figure 3).

Each of these idioms corresponds directly to a standard monad [18]. Now, the formlet idiom is just the composition of these three (Figure 5). The Compose module composes any two given idioms (Figure 4).

To work with a composed idiom, we need to be able to *lift* and *refine* the primitive operations between the component idioms. Given idioms F and G, we can *lift* any idiomatic computation of type $\alpha\, G.t$ to an idiomatic computation of type $\alpha\, G.t\ F.t$ using $F.pure$ or *refine* one of type $\alpha\, F.t$ to one of type $\alpha\, G.t\ F.t$ using $Compose(F)(G).refine$.

A combination of $N.pure$ and $AE.refine$ is used to lift the results of the $A.xml$ and $A.text$ operations to the *XmlWriter* idiom. The *tag* operation is lifted in a slightly different way as its third argument is an actual formlet: we *map* the

```
module Namer : sig
  include Idiom
  val next_name : string t
  val run : α t → α
end = struct
  type α t = int → α × int
  ...
end
```

```
module Environment : sig
  include Idiom
  type env = (string × string) list
  val lookup : string → string t
  val run : α t → env → α
end = struct
  type α t = env → α
  ...
end
```

```
module XmlWriter : sig
  include Idiom
  val text : string → unit t
  val xml : xml → unit t
  val tag : tag → attrs → α t → α t
  val run : α t → xml × α
end = struct
  type α t = xml × α
  ...
end
```

Fig. 3. Primitive idioms.

A.tag t ats operation over the formlet. The *run* operation simply runs each of the primitive *run* operations in turn. The *input* operation is the most interesting one. A fresh name is generated and used both to name an input element and for lookup in the environment once the form containing the element is submitted.

Note that, although the above idioms are in fact also monads, their composition, the formlet idiom, is not. (Recall that idioms are closed under composition while monads are not. Monad transformers recover some compositionality, but there is no combination of monad transformers that layers these effects in the right order.)

```
module Compose (F : Idiom) (G : Idiom) : sig
  include Idiom with type α t = α G.t F.t
  val refine : α F.t → α G.t F.t
end =
struct
  type α t = α G.t F.t
  let pure x = F.pure (G.pure x)
  let (⊗) f x = F.pure (⊗_G) ⊗_F f ⊗_F x
  let refine v = (F.pure G.pure) ⊗_F v
end
```

Fig. 4. Idiom composition.

```
module Formlet : FORMLET =
struct
  module AE = Compose (XmlWriter) (Environment)
  include Compose (NameGen) (AE)
  module N = NameGen module A = XmlWriter module E = Environment

  let xml x = N.pure (AE.refine (A.xml x))
  let text s = N.pure (AE.refine (A.text s))
  let tag t ats f = N.pure (A.tag t ats) ⊗N f
  let run v = let xml, collector = A.run (N.run v)
              in xml, E.run collector
  let input =
    N.pure (fun n → A.tag "input" [("name", n)] (A.pure (E.lookup n)))
      ⊗N N.next_name
end
```

Fig. 5. The formlet idiom, factored.

5 Syntax

The syntax of XML quasiquote expressions is given in Figure 6. A formlet expression, formlet q_f yields e, denotes a formlet whose structure is determined by the quasiquote q_f and whose outcome is calculated by the expression e (the *yields clause*), with variables bound according to the *formlet bindings* found in q_f. A *formlet binding*, of the form $\{f \Rightarrow x\}$, causes the variable x to be bound (within the yields clause) to the outcome computed by the formlet f. The rendering (form structure) of f takes the place of this binding within the rendering of the whole formlet expression.

The desugaring transformations are shown in Figure 7. The operation $(\cdot)^\circ$ desugars the formlet expressions in a program; it is a homomorphism on all syntactic forms except XML quasiquotes and formlet expressions. The auxiliary operations $[\![\cdot]\!]_\mathrm{x}$, $(\cdot)^\dagger$ and $[\![\cdot]\!]_\mathrm{f}$ are defined on quasiquotes and on nodes. Let r range over quasiquotes and nodes. The notation $r^\dagger$ denotes a pattern aggregating the sub-patterns of r. In an abuse of notation, we also let $r^\dagger$ denote the expression that reconstructs the value matched by the pattern. (Of course, we need to be somewhat careful in the OCaml implementation in order to make sure it is possible to reconstruct the value from the matched pattern.) The notation $[\![r]\!]_\mathrm{f}$ denotes a formlet aggregating the sub-formlets of r.

As well as pure and apply, the operations *text*, *xml* and *tag* are used for lifting raw XML into formlets.

As a simple example of desugaring, consider the definition of the *input_int* formlet used earlier:

```
let input_int : int formlet =
  formlet <#>{input ⇒ i}</#> yields int_of_string i
```

Under the translation given in Figure 7, the body becomes

```
pure (fun i → int_of_string i) ⊗ (pure (fun i → i) ⊗ input)
```

Expressions

$$\begin{array}{rcll} e & ::= & \cdots \mid q_{\mathrm{x}} & \text{XML} \\ & & \mid \mathsf{formlet}\ q_{\mathrm{f}}\ \mathsf{yields}\ e & \text{formlet} \end{array}$$

XML quasiquotes

$$\begin{array}{rcll} n & ::= & s \mid \texttt{\{}e\texttt{\}} \mid \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} & \text{node} \\ q_{\mathrm{x}} & ::= & \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} \mid \texttt{<\#>}n_1 \ldots n_k\texttt{</\#>} & \text{quasiquote} \end{array}$$

Formlet quasiquotes

$$\begin{array}{rcll} n & ::= & s \mid \texttt{\{}e\texttt{\}} \mid \texttt{\{}f \Rightarrow p\texttt{\}} \mid \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} & \text{node} \\ q_{\mathrm{f}} & ::= & \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>} \mid \texttt{<\#>}n_1 \ldots n_k\texttt{</\#>} & \text{quasiquote} \end{array}$$

Meta variables

e	expression	f	formlet-type expression	t	tag
p	pattern	s	string	as	attribute list

Fig. 6. Quasiquote syntax.

$$\begin{aligned} (q_{\mathrm{x}})^{\circ} &= [\![q_{\mathrm{x}}]\!]_{\mathrm{x}} \\ (\mathsf{formlet}\ q\ \mathsf{yields}\ e)^{\circ} &= \mathit{pure}\,(\mathsf{fun}\ q^{\dagger} \to e^{\circ})\ \otimes\ [\![q]\!]_{\mathrm{f}} \end{aligned}$$

$$\begin{aligned} [\![s]\!]_{\mathrm{x}} &= \mathit{xml\_text}\ s \\ [\![\texttt{\{}e\texttt{\}}]\!]_{\mathrm{x}} &= e^{\circ} \\ [\![\texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}]\!]_{\mathrm{x}} &= \mathit{xml\_tag}\ t\ as\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]_{\mathrm{x}} \\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]_{\mathrm{x}} &= [\![n_1]\!]_{\mathrm{x}}\ @\ \ldots\ @\ [\![n_k]\!]_{\mathrm{x}} \end{aligned}$$

$$\begin{aligned} [\![s]\!]_{\mathrm{f}} &= \mathit{text}\ s \\ [\![\texttt{\{}e\texttt{\}}]\!]_{\mathrm{f}} &= \mathit{xml}\ e^{\circ} \\ [\![\texttt{\{}f \Rightarrow p\texttt{\}}]\!]_{\mathrm{f}} &= f^{\circ} \\ [\![\texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}]\!]_{\mathrm{f}} &= \mathit{tag}\ t\ as\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]_{\mathrm{f}} \\ [\![\texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}]\!]_{\mathrm{f}} &= \mathit{pure}(\mathsf{fun}\ n_1^{\dagger} \ldots n_k^{\dagger} \to (n_1^{\dagger}, \ldots, n_k^{\dagger})) \otimes [\![n_1]\!]_{\mathrm{f}}\ \cdots\ \otimes [\![n_k]\!]_{\mathrm{f}} \end{aligned}$$

$$\begin{aligned} s^{\dagger} &= \texttt{()} \\ \texttt{\{}e\texttt{\}}^{\dagger} &= \texttt{()} \\ \texttt{\{}f \Rightarrow p\texttt{\}}^{\dagger} &= p \\ \texttt{<}t\ as\texttt{>}n_1 \ldots n_k\texttt{</}t\texttt{>}^{\dagger} &= n_1^{\dagger} \ldots n_k^{\dagger} \\ \texttt{<\#>}n_1 \ldots n_k\texttt{</\#>}^{\dagger} &= n_1^{\dagger} \ldots n_k^{\dagger} \end{aligned}$$

Fig. 7. Desugaring XML and formlets.

We can use the idiom laws (and η-reduction) to simplify the output a little, giving the following semantically-equivalent code:

```
pure int_of_string ⊗ input
```

More opportunities for optimisation arise with larger expressions. For example, the $\otimes$ operator of the formlet idiom may be written as follows using syntactic sugar:

```
let (⊗) : (α → β) formlet → α formlet → β formlet = fun f p →
  formlet <#>{f ⇒ g}{p ⇒ q}</#> yields g q
```

Under the desugaring transformation, the body becomes

```
(pure (fun (g, q) → g q)) ⊗ (pure (fun g q → (g, q)) ⊗ f ⊗ p)
```

which, under the idiom laws, is equivalent to $f \otimes p$.

As a richer example, recall the *date* formlet from Section 3. It desugars to the following code:

```
let date : date formlet =
  pure (fun ((), month, (), day, ()) → make_date month day)
  ⊗ (tag "div" []
       (pure (fun () month () day () → ((), month, (), day, ()))
        ⊗ text "\n      Month : " ⊗ input_int
        ⊗ text "\n      Day   : " ⊗ input_int ⊗ text "\n    "))
```

We could easily optimise this code by deforesting the extra units from the body of the inner *pure* and from the arguments to the function in the outer *pure*. One thing we cannot do is avoid the rebinding of *month* and *day*. In Section 6.3 we outline how to fix this limitation.

6 Extensions

The formlet abstraction is robust, as we can show by extending it in several independent ways.

6.1 XHTML validation

The problem of statically enforcing validity of HTML and indeed XML is well-studied [4, 25, 12, 20]. Such schemes are essentially orthogonal to the work presented here: we can incorporate a type system for XML with little disturbance to the core formlet abstraction.

Of course, building static validity into the type system requires that we have a whole family of types for HTML rather than just one. For instance (as in Elsman and Larsen's system [8]), we might have one type for `block` entitites and another for `inline` entities, or (as in XDuce [12]) we might even have a different type for every tag.

Fortunately, it is easy to push the extra type parameters through our formlet construction. The key component that needs to change is the *XmlWriter* idiom.

As well as the value type, this now needs to be parameterised over the XML type. The construction we need is the idiom analogue of effect-indexed monads [28].

In OCaml, we define an *indexed idiom* as follows:

```
module type XIdiom = sig
  type (ψ, α) t
  val pure : α → (ψ, α) t
  val (⊗) : (ψ, α → β) t → (ψ, α) t → (ψ, β) t
end
```

(For the indexed XML writer idiom the parameter ψ is the XML type.) Like idioms, indexed idioms satisfy the four laws given in Section 4. They can be pre- and post-composed with other idioms to form new indexed idioms. Pre-composing the name generation idiom with the indexed XML writer idiom pre-composed with the environment idiom gives us an indexed formlet idiom.

As a proof of concept, we have implemented a prototype of formlets with XML typing in OCaml using Elsman and Larsen's encoding of a fragment of XHTML 1.0 [8]. It uses phantom types to capture XHTML validity constraints.

6.2 Input validation

A common need in form processing is validating user input: on submission, we should ensure that the data is well-formed, and if not, re-display the form to the user (with error messages) until well-formed data is submitted.

Formlets extend to this need if we incorporate additional idioms for error-checking and accumulating error messages and add combinators *satisfies* and *err*, which add to a formlet, respectively, an assertion that the outcome must satisfy a given predicate and an error message to be used when it does not. Any time the continuation associated with a formlet is invoked, the outcome is sure to satisfy the validation predicate(s).

The need to re-display a page upon errors also requires additional mechanics. Instead of simply attaching a continuation to a formlet and rendering it to HTML, the formlet continuation now needs to have a complete page context available to it, in case it needs to redisplay the page. To facilitate this, we add a new syntactic form, which associates formlets with their continuations *in the context of* a larger page.

Extending with input validation adds some complexity to the implementation, so we omit details here. We have implemented it in the Links version of formlets and provide details in a technical report [7].

6.3 Multi-holed contexts

The presentation of formlets we have given in this paper relies on lifting the *tag* constructor from the *XmlWriter* idiom into the *Formlet* idiom. As illustrated by the desugaring of the date example in Section 5 this makes it difficult to separate the raw XML from the semantic content of formlets and requires nested formlet values to be rebound.

Apart from obfuscating the code, this rebinding is inefficient [for the final version of the paper we will provide benchmarks]. By adapting the formlet datatype to accumulate a list of XML values rather than a single XML value, and replacing *tag* with a general operation for plugging the accumulated list into a multi-holed context *plug*, we obtain a more efficient formlet implementation that does provide a separation between the raw XML and the semantic content. Further, this leads to a much more direct desugaring transformation. For example, the desugared version of the date example becomes:

```
let date : (_, date) NFormlet.t =
  plug (tag "div" []
          (text "\n      Month : " ++ hole ++
           text "\n      Day   : " ++ hole ++
           text "\n    " ++ empty))
       (pure (fun month day → make_date month day) ⊗ input_int ⊗ input_int)
```

Statically typing *plug* in OCaml requires some ingenuity. Using phantom types, we encode the number of holes in a context, or the number of elements in a list, as the difference between two type-level Peano numbers [15]. As with XHTML typing the key component that needs to change is the *XmlWriter* idiom. This now needs to be parameterised over the number of XML values in the list it accumulates. The construction we need is the idiom analogue of parameterised monads [1]. In OCaml, we define a *parameterised idiom* as follows:

```
module type PIdiom = sig
  type (μ, ν, α) t
  val pure : α → (μ, ν, α) t
  val (⊗) : (μ, ν, α → β) t → (σ, μ, α) t → (σ, ν, β) t
end
```

(For the parameterised XML writer idiom the parameters μ and ν encode the length of the list of XML values as $\nu - \mu$.) Like idioms, and indexed idioms, parameterised idioms satisfy the four laws given in Section 4. They can be pre- and post-composed with other idioms to form new parameterised idioms. Pre-composing the name generation idiom with the parameterised XML writer idiom pre-composed with the environment idiom gives a parameterised formlet idiom.

We have implemented a prototype of formlets with a multi-holed plugging operation in OCaml. Statically-typed multi-holed contexts can be combined with statically typed XHTML [15]. Lifting the result to idioms gives either an *indexed parameterised idiom* — that is, an idiom with an extra type parameter for the XML type and two extra type parameters for the number of XML values in the accumulated list — or, by attaching the XML type to both of the other type parameters, a parameterised idiom.

6.4 Other extensions

These are by no means the only useful extensions to the basic formlet abstraction. For example, we might wish to translate validation code to JavaScript to run on the client [11], or enforce separation between those portions of the program that

deal with presentation and those that treat application-specific computation, a common requirement in large web projects. Either of these may be combined with the formlet abstraction without injury to the core design presented here.

7 Related work

The WASH, iData and WUI frameworks all support aspects of the form abstraction we have presented. WUI, in fact, meets all of the goals mentioned in the introduction. Underlying all these systems is the essential mode of form abstraction that we have presented, although they vary richly in their feature sets and limitations.

WASH The WASH/CGI Haskell framework [26] supports a variety of web application needs, including forms with some abstraction. WASH supports user-defined types as the result of an individual form field, through defining a `Read` instance, which parses the type from a string. It also supports aggregating data from multiple fields using a suite of tupling constructors, but it does not allow arbitrary calculations from these multiple fields into other data types, such as our abstract Date type. In particular, the tupling constructors still expose the structure of the form fields, preventing true abstraction. For example, given a one-field component, a programmer cannot modify it to consist of two fields without also changing all the uses of the component.

iData The iData framework [22] supports a high degree of form abstraction, calling its abstractions *iData*. An iData can be defined at any type, thus it supports arbitrary packaging of form outcomes. However, each iData on a page must be given a particular name, which exposes the programmer to the number of instances of a particular iData.

WUI The WUI (*Web User Interface*) library [10, 11] implements form abstractions for the functional logic programming language Curry. Here the basic units are called WUIs. WUIs enforce an assumption that each WUI of type α should accept a value of type α as well as generate one; this input value would model the default or current value for the component. Thus a $WUI\ \alpha$ is equivalent, in our setting, to a value of type $\alpha \to \alpha\ Formlet$.

References

1. Robert Atkey. Parameterised notions of computation. In *MSFP*, 2006.
2. Vincent Balat. Ocsigen: typing web interaction with objective caml. In *ML Workshop '06*, pages 84–94, 2006.
3. Nick Benton, John Hughes, and Eugenio Moggi. Monads and effects. In *Applied Semantics: Advanced Lectures*, volume 2395 of *LNCS*, pages 42–122, 2002.
4. Claus Brabrand, Anders Møller, and Michael I. Schwartzbach. Static validation of dynamically generated HTML. In *PASTE*, pages 38–45, 2001.
5. Aske Simon Christensen, Anders Møller, and Michael I. Schwartzbach. Extending Java for high-level web service construction. *ACM Trans. Program. Lang. Syst.*, 25(6):814–875, 2003.

6. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. Links: web programming without tiers. In *FMCO 2006*, volume 4709 of *LNCS*, pages 266–296, 2007.
7. Ezra Cooper, Sam Lindley, Philip Wadler, and Jeremy Yallop. An idiom's guide to formlets. Technical Report EDI-INF-RR-1263, School of Informatics, University of Edinburgh, 2008.
8. Martin Elsman and Ken Friis Larsen. Typing XHTML web applications in ML. In *PADL '04*, pages 224–238, 2004.
9. Paul T. Graunke, Shriram Krishnamurthi, Steve Van Der Hoeven, and Matthias Felleisen. Programming the web with high-level programming languages. In *ESOP '01*, pages 122–136, 2001.
10. Michael Hanus. Type-oriented construction of web user interfaces. In *PPDP '06*, pages 27–38, 2006.
11. Michael Hanus. Putting declarative programming into the web: Translating Curry to JavaScript. In *PPDP'07*, pages 155–166, 2007.
12. Haruo Hosoya and Benjamin C. Pierce. XDuce: A statically typed XML processing language. *ACM Trans. Internet Techn.*, 3(2):117–148, 2003.
13. John Hughes. Generalising monads to arrows. *Sci. Comput. Program.*, 37(1-3):67–111, 2000.
14. Lift website, March 2008. `http://liftweb.net/`.
15. Sam Lindley. Many holes in Hindley-Milner. Technical Report EDI-INF-RR-1262, School of Informatics, University of Edinburgh, 2008.
16. Sam Lindley, Philip Wadler, and Jeremy Yallop. Idioms are oblivious, arrows are meticulous, monads are promiscuous. In Venanzio Capretta and Conor McBride, editors, *MSFP '08*, Reykjavik, Iceland., 2008. To appear.
17. Conor McBride. Idioms, 2005. Presented at SPLS June 2005 `http://www.macs.hw.ac.uk/~trinder/spls05/McBride.html`.
18. Conor McBride and Ross Paterson. Applicative programming with effects. *Journal of Functional Programming*, 18(1), 2008.
19. Eugenio Moggi. Computational lambda-calculus and monads. In *LICS '89*, pages 14–23, 1989.
20. Anders Møller and Michael I. Schwartzbach. The design space of type checkers for XML transformation languages. In *ICDT '05*, January 2005.
21. PHP Hypertext Preprocessor, March 2008. `http://www.php.net/`.
22. Rinus Plasmeijer and Peter Achten. iData for the world wide web: Programming interconnected web forms. In *FLOPS '06*, pages 242–258, 2006.
23. Ruby on Rails website, March 2008. `http://www.rubyonrails.org/`.
24. Steve Strugnell. Creating linksCollab: an assessment of Links as a web development language. Bachelor thesis, The University of Edinburgh `http://groups.inf.ed.ac.uk/links/papers/undergrads/steve.pdf`, 2008.
25. Peter Thiemann. A typed representation for HTML and XML documents in Haskell. *J. Funct. Program.*, 12(4&5):435–468, 2002.
26. Peter Thiemann. An embedded domain-specific language for type-safe server-side web scripting. *ACM Trans. Inter. Tech.*, 5(1):1–46, 2005.
27. Philip Wadler. Monads for functional programming. In *Advanced Functional Programming '95*, volume 925 of *LNCS*, pages 24–52. 1995.
28. Philip Wadler. The marriage of effects and monads. In *ICFP*, pages 63–74, 1998.